TESLA
GPU Computing
Libraries, Directives, CUDA
Ways to Accelerate on GPU

Application

Libraries
Directives
Programming Languages

Easiest Approach for 2x to 10x Acceleration
Maximum Performance
C, C++, Fortran Language Integration with CUDA Parallel Programming Model

**GPU Computing Applications**

<table>
<thead>
<tr>
<th>Libraries and Middleware</th>
<th>Libraries and Middleware</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuFFT, cuBLAS, cuRAND, cuSPARSE</td>
<td>LAPACK, CULA, MAGMA</td>
</tr>
<tr>
<td>NPP, cuDPP, Thrust</td>
<td>SVM, OpenCurrent</td>
</tr>
<tr>
<td>PhysX, Video, OptiX, Ray, Rendering, RealityServer</td>
<td>iray, Rendering, RealityServer</td>
</tr>
<tr>
<td>MATLAB, Mathematica</td>
<td></td>
</tr>
</tbody>
</table>

| C++ | C | Fortran | Java Python Wrappers | Direct Compute | OpenCL™ |

**NVIDIA GPU**
CUDA Parallel Computing Architecture
GPU Accelerated Libraries

“Drop-in” Acceleration for Your Applications
CUDA Math Libraries

High performance math routines for your applications:
- cuFFT – Fast Fourier Transforms Library
- cuBLAS – Complete BLAS Library
- cuSPARSE – Sparse Matrix Library
- cuRAND – Random Number Generation (RNG) Library
- NPP – Performance Primitives for Image & Video Processing
- Thrust – Templated Parallel Algorithms & Data Structures
- math.h - C99 floating-point Library

- Included in the CUDA Toolkit (free download)
  - www.nvidia.com/getcuda
- For more information on CUDA libraries:
cuFFT: Multi-dimensional FFTs

New in CUDA 4.0

Significant performance improvements in:
- double precision radix 2, 3, 5 and 7
- 2D/3D sizes that contain prime factors larger than 7

Flexible input and output data layouts*
- Similar to the FFTW “Advanced Interface”
- Eliminates extra data transposes and copies

* Only supported for complex-to-complex transforms in this release

\[ F(x) = \sum_{n=0}^{N-1} f(n)e^{-j2\pi(x_n/N)} \]

\[ f(n) = \frac{1}{N} \sum_{n=0}^{N-1} F(x)e^{j2\pi(x_n/N)} \]
FFTsd up to 10x Faster than MKL

1D used in audio processing and as a foundation for 2D and 3D FFTs

- MKL 10.1r1 on Intel Quad Core i7-940 1333, 2.93Ghz
- cuFFT 4.0 on Tesla C2070, ECC on
- Performance measured for ~16M total elements, split into batches of transforms of the size on the x-axis
2D/3D primes now use Bluestein Algorithm

Significant performance improvement for 2D and 3D transform sizes

- MKL 10.1r1 on Intel Quad Core i7-940 1333, 2.93Ghz
- cuFFT 4.0 on C2070, ECC on

![Graph showing performance improvement](image-url)

- cuFFT Single Precision 2D
- cuFFT 4.0
- cuFFT 3.2
- MKL

* MKL 10.1r1 on Intel Quad Core i7-940 1333, 2.93Ghz
* cuFFT 4.0 on C2070, ECC on
cuBLAS: Dense Linear Algebra on GPUs

- Complete BLAS implementation plus useful extensions
  - Supports all 152 standard routines for single, double, complex, and double complex

New in CUDA 4.0
- New API
  - Facilitates multi-GPU programming
  - Thread-safe
  - More routines provide parallelism using streams
  - Previous “legacy” API still supported out-of-the-box
- Rewrote documentation from scratch
- Performance improvements
  - Ex: ZGEMM performance improved 10% on Fermi (325 GFLOPS peak on C2050)
cuBLAS Level 3 Performance

Up to ~800 GFLOPS and ~17x speedup over MKL

* 4Kx4K matrix size
* cuBLAS 4.0, Tesla C2050 (Fermi), ECC on
* MKL 10.2.3, 4-core Corei7 @ 2.66Ghz
ZGEMM Performance vs. Matrix Size

Up to **8x** speedup over MKL

- **97% of peak perf. (1024x1024)**
- **85% of peak perf. (512x512)**
- **38% of peak perf. (512x512)**
- **80% of peak perf. (1024x1024)**

Performance may vary based on OS version and motherboard configuration

* cuBLAS 4.0, Tesla C2050 (Fermi), ECC on
* MKL 10.2.3, 4-core Corei7 @ 2.66Ghz
cuSPARSE: Sparse linear algebra routines

- Conversion routines for dense, COO, CSR and CSC formats
- Optimized sparse matrix-vector multiplication for CSR
- New Sparse Triangular Solve CUDA 4.0
  - API optimized for common iterative solve algorithms

\[
\begin{bmatrix}
  y_1 \\
  y_2 \\
  y_3 \\
  y_4 \\
\end{bmatrix}
= \alpha \begin{bmatrix}
  1.0 & & & \\
  2.0 & 3.0 & & \\
  5.0 & 6.0 & 7.0 & \\
\end{bmatrix}
\begin{bmatrix}
  1.0 \\
  2.0 \\
  3.0 \\
  4.0 \\
\end{bmatrix}
+ \beta \begin{bmatrix}
  y_1 \\
  y_2 \\
  y_3 \\
  y_4 \\
\end{bmatrix}
\]
cuSPARSE is up to 6x Faster than MKL

Sparse Matrix x Dense Vector

Performance may vary based on OS version and motherboard configuration

* cuSPARSE 4.0, NVIDIA C2050 (Fermi), ECC on
* MKL 10.2.3, 4-core Corei7 @ 3.07GHz
Up to 35x faster with 6 Dense Vectors

Useful for block iterative solve schemes

Performance may vary based on OS version and motherboard configuration

* cuSPARSE 4.0, NVIDIA C2050 (Fermi), ECC on
* MKL 10.2.3, 4-core Corei7 @ 3.07GHz
cuRAND: Random Number Generation

New in CUDA 4.0
- Scrambled and 64-bit Sobol’
- Log-normal distribution
- New parallel ordering supports faster XORWOW initialization
- Results of CURAND generators against standard statistical test batteries are reported in documentation
cuRAND Performance

cuRAND 64-bit Scrambled Sobol’ 8x faster than MKL 32-bit plain Sobol’

Performance may vary based on OS version and motherboard configuration

* CURAND 4.0, NVIDIA C2050 (Fermi), ECC on
NVIDIA Performance Primitives

Up to 40x speedups

- Arithmetic, Logic, Conversions, Filters, Statistics, etc.
  - ~420 image functions (+70 in 4.0)
  - ~500 signal functions (+400 in 4.0)
- Majority of primitives 5x to 10x faster than analogous routines in Intel IPP

*NPP 4.0, NVIDIA C2050 (Fermi)
* IPP 6.1, Dual Socket Core™ i7 920 @ 2.67GHz
Thrust: CUDA C++ Template Library

- Added to CUDA Toolkit as of CUDA 4.0
  - Also available on Google Code

- Template library for CUDA
  - Host and Device Containers that mimic the C++ STL
  - Optimized algorithms for sort, reduce, scan, etc.
  - OpenMP backend for portability

- Allows applications and prototypes to be built quickly
Thrust Algorithm Performance

Various Algorithms (32M int.)
Speedup compared to C++ STL

Sort (32M samples)
Speedup compared to C++ STL

* Thrust 4.0, NVIDIA Tesla C2050 (Fermi) * Core i7 950 @ 3.07GHz
CUDA math.h is industry proven, high performance, high accuracy

- **Basic**: +, *, /, 1/, sqrt, FMA (all IEEE-754 accurate for float, double, all rounding modes)
- **Exponentials**: exp, exp2, log, log2, log10, ...
- **Trigonometry**: sin, cos, tan, asin, acos, atan2, sinh, cosh, asinh, acosh, ...
- **Special functions**: lgamma, tgamma, erf, erfc
- **Utility**: fmod, remquo, modf, trunc, round, ceil, floor, fabs, ...
- **Extras**: rsqrt, rcbtrt, exp10, sinpi, sincos, cospi, erfinv, erfcinv, ...

- **Performance improvements in CUDA 4.0**
  - Double-precision /, rsqrt(), erfc(), & sinh() are all >~30% faster on Fermi
- **Added cospi() to CUDA 4.0**
MAGMA

- LAPACK from Jack Dongarra’s group at UTK
- Hybrid algorithms use CPU and GPU
- Selected routines ported to multi-gpu
- [http://icl.cs.utk.edu/magma/software](http://icl.cs.utk.edu/magma/software)
CULA

- Linear algebra routines, LAPACK from EMPHotonics
- Hybrid CPU/GPU algorithms
- Currently single GPU only
- [http://www.culatools.com](http://www.culatools.com)
Directives: Simple Hints

Add hints to code
On-ramp to parallel computing
Compiler does heavy lifting of parallelizing code
Works on multicore CPUs & many core GPUs

CPU

GPU

Original C/Fortran code

main() {
  ...
  <serial code>
  ...
  #pragma acc region
  {
    <compute intensive code>
  }
  ...
}
Directives: Add A Few Lines of Code

OpenMP

CPU

```
main() {
  double pi = 0.0; long i;

  #pragma omp parallel for reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

  printf("pi = %f\n", pi/N);
}
```

Directives

CPU

```
main() {
  double pi = 0.0; long i;

  #pragma omp parallel for reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

  printf("pi = %f\n", pi/N);
}
```

GPU

```
main() {
  double pi = 0.0; long i;

  #pragma omp acc_region_loop
  #pragma omp parallel for reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

  #pragma omp end acc_region_loop
  printf("pi = %f\n", pi/N);
}
```
Small Effort. Real Impact.

Large Oil Company
3x in 7 days
Solving billions of equations iteratively for oil production at world’s largest petroleum reservoirs

Univ. of Houston
Prof. M.A. Kayali
20x in 2 days
Studying magnetic systems for innovations in magnetic storage media and memory, field sensors, and biomagnetism

Uni. Of Melbourne
Prof. Kerry Black
65x in 2 days
Better understand complex reasons by lifecycles of snapper fish in Port Phillip Bay

Ufa State Aviation
Prof. Arthur Yuldashev
7x in 4 Weeks
Generating stochastic geological models of oilfield reservoirs with borehole data

GAMESS-UK
Dr. Wilkinson, Prof. Naidoo
10x
Used for various fields such as investigating biofuel production and molecular sensors.
Legacy Codes Now Faster on CPUs & GPUs

**PROBLEM**
Numerous codes written decades ago

**SOLUTION**
Minor changes using directives with GPUs

With compiler directives, we are making minor changes to codes that are decades old to leverage the computational capabilities of NVIDIA GPUs. Some effort is required but the payoff is tremendous. We’ve seen some codes run 2x faster even on CPU-only platforms due to just improved parallel code.

Bronson Messer
Adjunct Asst. Professor at University of Tennessee
R&D Staff at NCCS
2x in 4 Weeks. Guaranteed.

Free 30 day trial license to PGI Accelerator*

Tools for quick ramp

www.nvidia.com/2xin4weeks

*Limit 1000 developers
OpenACC: Open Programming Standard for Parallel Computing
Easy, Fast, Portable

“OpenACC will enable programmers to easily develop portable applications that maximize the performance and power efficiency benefits of the hybrid CPU/GPU architecture of Titan.”

OpenACC is a technically impressive initiative brought together by members of the OpenMP Working Group on Accelerators, as well as many others. We look forward to releasing a version of this proposal in the next release of OpenMP.”

Buddy Bland
Titan Project Director
Oak Ridge National Lab

Michael Wong
CEO, OpenMP Directives Board
CUDA Tools Overview

- **CUDA Parallel Compute Engines inside GPU**
- **CUDA Support in Kernel Level Driver**
- **Applications Using DirectX**
  - DirectX 11 Compute
  - HLSL
- **Applications Using OpenCL**
  - OpenCL C
- **Applications Using the CUDA Driver API**
  - C for CUDA
- **Applications Using C, C++, Fortran, Java, Python, ...**
  - C for CUDA
- **CUDA Driver**
  - PTX (ISA)
  - C Runtime for CUDA

Device-level APIs

Language Integration
Parallelize using CUDA Programming Model

Rest of Sequential CPU Code

Critical Functions

Application Code

CPU

GPU
1. Copy input data from CPU memory to GPU memory
Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
CUDA Compilation Flowchart

C/C++ CUDA Application

NVCC

PTX Code

PTX to Target Compiler

Target code

G80

…

Fermi

CPU Code
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);

__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nbblocks = (n + 255) / 256;
saxpy_parallel<<<nbblocks, 256>>>(n, 2.0, x, y);
void serial_function(...) {
    ...
}
void other_function(int ...) {
    ...
}
__global__ void saxpy_parallel(int n, float a, float *x, float *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
void main() {
    float x;
    saxpy_parallel<<<nbloks, 256>>>(n, 2.0, x, y);
    ...
}
CUDA Programming Effort / Performance

These things are REALLY fast

- **Performance (gflops)**
  - Matlab: 0.3, 0.5
  - C/SSE: 9.0, 10.0
  - PS3: 30.0
  - GT200: 110.0

- **Development Time (hours)**
  - Matlab: 10.0
  - C/SSE: 110.0
  - PS3: 330.0
  - GT200: 10.0

Source: MIT CUDA Course
OpenCL & CUDA

- NVIDIA Supports all GPU programming languages
  - CUDA C/C++ is our platform of innovation
- Language integration, tools and libraries yield higher productivity
  - CUDA C Runtime has them today, OpenCL device driver API does not
- OpenCL designed for “write once, optimize everywhere”
  - Kernels must be optimized for each processor architecture
- NVIDIA is still the leading vendor delivering support for OpenCL
  - Conformant in June 2009, in production drivers since September

NVIDIA released the first OpenCL v1.0 conformant driver for Windows and Linux to thousands of developers in June 2009
Intro to CUDA C

Memory Management
Memory Spaces

CPU and GPU have separate memory spaces
- Data is moved across PCIe bus
- Use functions to allocate/set/copy memory on GPU
  - Very similar to corresponding C functions

Pointers are just addresses
- Can’t tell from the pointer value whether the address is on CPU or GPU
- Must exercise care when dereferencing:
  - Dereferencing CPU pointer on GPU will likely crash
  - Dereferencing GPU pointer on CPU will likely crash
GPU Memory Allocation / Release

Host (CPU) manages device (GPU) memory

- `cudaMalloc (void ** pointer, size_t nbytes)`
- `cudaMemset (void * pointer, int value, size_t count)`
- `cudaFree (void* pointer)`

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**)&d_a, nbytes );
cudaMemset( d_a, 0, nbytes );
cudaFree(d_a);
```

**Note:** Device memory from GPU point of view is also referred to as global memory.
Data Copies

cudA::memcpy( void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);

- returns after the copy is complete
- blocks CPU thread until all bytes have been copied
- doesn’t start copying until previous CUDA calls complete

enum cudaMemcpyKind

- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToDevice

Non-blocking memcopies are provided
Code Walkthrough 1

- Allocate CPU memory for $n$ integers
- Allocate GPU memory for $n$ integers
- Initialize GPU memory to 0s
- Copy from GPU to CPU
- Print the values
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers
```c
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMemcpy( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a ) {
        printf("couldn't allocate memory\n"); return 1;
    }
```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMemcpy( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a ) {
        printf("couldn't allocate memory\n"); return 1;
    }

    cudaMemcpy( d_a, h_a, num_bytes, cudaMemcpyDeviceToHost );
}
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a ) {
        printf("couldn't allocate memory\n"); return 1;
    }

    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int i=0; i<dimx; i++)
        printf("%d ", h_a[i] );

    free( h_a );
    cudaFree( d_a );

    return 0;
}
Intro to CUDA C

Basic Kernels
and Execution on the GPU
CUDA Programming Model

- Parallel code (kernel) is launched and executed on a device by many threads
- Threads are grouped into thread blocks
- Parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in thread and block ID variables
Thread Hierarchy

- Threads launched for a parallel section are partitioned into thread blocks
  - Grid = all blocks for a given launch
- Thread block is a group of threads that can:
  - Synchronize their execution
  - Communicate via shared memory
IDs and Dimensions

Threads
- 3D IDs, unique within a block

Blocks
- 2D IDs, unique within a grid

Dimensions set at launch time
- Can be unique for each grid

Built-in variables
- threadIdx, blockIdx
- blockDim, gridDim

(Continued)
IDs and Dimensions

Threads
- 3D IDs, unique within a block

Blocks
- 2D IDs, unique within a grid

Dimensions set at launch time
- Can be unique for each grid

Built-in variables
- threadIdx, blockIdx
- blockDim, gridDim
Code executed on GPU

C function with some restrictions:
- Can only access GPU memory (0-copy is the exception)
- No variable number of arguments
- No static variables
- No recursion

Must be declared with a qualifier:
- __global__: launched by CPU, cannot be called from GPU must return void
- __device__: called from other GPU functions, cannot be launched by the CPU
- __host__: can be executed by CPU
- __host__ and __device__ qualifiers can be combined
  - sample use: complex mathematical functions
Code Walkthrough 2

- Build on Walkthrough 1
- Write a kernel to initialize integers
- Copy the result back to CPU
- Print the values
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
Launching Kernels on GPU

Launch parameters

- grid dimensions (up to 2D), dim3 type
- thread-block dimensions (up to 3D), dim3 type
- shared memory: number of bytes per block
  for extern smem variables declared without size
  Optional, 0 by default
- stream ID
  Optional, 0 by default

```c
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block, 0, 0>>>(...);
kernel<<<32, 512>>>(...);
```
```c
#include <stdio.h>

__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a, *h_a; // device and host pointers
    h_a = (int*)malloc(num_bytes);
    cudaMemcpy( (void**)&d_a, num_bytes );
    if( 0==h_a || 0==d_a )  {
        printf("couldn't allocate memory\n"); return 1;
    }

    cudaMemcpy( d_a, num_bytes );

    dim3 grid, block;
    block.x = 4;
    grid.x  = dimx / block.x;

    kernel<<<grid, block>>>( d_a );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int i=0; i<dimx; i++)
        printf("%d ", h_a[i] );

    free( h_a );
    cudaFree( d_a );
    return 0;
}
```
Kernel Variations and Output

```c
__global__ void kernel( int *a )
{
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    a[id] = 7;
}
```

Output: `7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7`

```c
__global__ void kernel( int *a )
{
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    a[id] = blockIdx.x;
}
```

Output: `0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3`

```c
__global__ void kernel( int *a )
{
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    a[id] = threadIdx.x;
}
```

Output: `0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3`
GPU Kernel Execution

How the HW executes kernels

- GPU consists of multiple cores (Streaming Multiprocessors, up to 30)
- Blocks are launched on SMs
- Each SM can have multiple concurrent blocks executing
- Once a block is started it will not migrate to another SM
Blocks Must Be Independent

Any possible interleaving of blocks should be valid
- presumed to run to completion without pre-emption
- can run in any order
- can run concurrently OR sequentially

Blocks may coordinate but not synchronize
- shared queue pointer: OK
- shared lock: BAD … any dependence on order easily deadlocks

Independence requirement gives scalability
Blocks Must Be Independent

Facilitates scaling of the same code across many devices
Questions?